

MPP-2006-46
hep-ph/0605049

SUSY Les Houches Accord 2 I/O made easy

T. Hahn
Max-Planck-Institut für Physik
Föhringer Ring 6, D-80805 Munich, Germany

Oct 18, 2006

Abstract

A library for reading and writing data in the SUSY Les Houches Accord 2 format is presented. The implementation is in native Fortran 77. The data are contained in a single array conveniently indexed by preprocessor statements.

1 Introduction

The original SUSY Les Houches Accord [1] (SLHA1 in the following) has standardized and significantly simplified the exchange of MSSM input and output parameters between such disparate applications as spectrum calculators and event generators. Meanwhile, agreement has been reached also about the encoding of many extensions of the MSSM which has led to a preliminary SLHA2 document [2].

While the SLHA specifications include the precise formats for Fortran I/O, it is nevertheless not entirely straightforward to read or write a file in SLHA format. The present library provides the user with simple routines to read and write files in SLHA format, as well as a few utility routines. One thing the library does not do is modify the numbers, which means there is no routine to compute, say, a particular quantity at a new scale. The data structures and subroutines are set up such that only very few changes are necessary when upgrading from the SLHALib 1 [3].

Sect. 2 describes the organization of the data structures, Sect. 3 gives the reference information for the library routines, Sect. 4 shows the usage in some examples, Sect. 5 contains download and build instructions, and Sect. 6 summarizes.

2 Data structures

The SLHA library is written in Fortran 77. All routines operate on a double complex array, `slhadata`, which is about the simplest conceivable data format for this purpose in Fortran. For convenience of use, this array is accessed via preprocessor statements, so the

user never needs to memorize any actual indices for the `slhadata` array. A file containing the preprocessor definitions must thus be included.

The `slhadata` array consists of a ‘static’ part containing the information from SLHA BLOCK sections and a ‘dynamic’ part containing the information from SLHA DECAY sections. The static part is indexed by preprocessor variables defined in `SLHA.h`, the dynamic part is accessed through the `SLHANewDecay`, `SLHAFindDecay`, `SLHAAddDecay`, `SLHAGetDecay`, and `SLHADecayTable` functions and subroutines (see Sect. 3).

In addition, descriptive names for the PDG codes of the particles are declared in `PDG.h`. These are needed e.g. to access the decay information.

2.1 SLHA blocks

The explicit indexing of the `slhadata` need not (and should not) be done by the user. Rather, the members of the SLHA data structure are accessed through preprocessor variables. Tables 1–10 list the preprocessor variables defined in `SLHA.h` which follow closely the definition of the Accord. Note that preprocessor symbols are case sensitive. On the downside, there is no way to guard against out-of-range indices, not even with compiler flags. This is because the preprocessor has no such checks and the compiler cannot determine *a posteriori* whether the single index it sees addresses the ‘right’ part of the array.

As far as there is overlap, the names for the block members have been chosen similar to the ones used in the MSSM model file of *FeynArts* [4]. Following is a list of common index conventions. This is only for a rough orientation: the actual indices and their ranges are always given explicitly in the Tables.

$t = 1 \dots 4$	(s)fermion type: 1 = (s)neutrinos, 2 = isospin-down (s)leptons, 3 = isospin-up (s)quarks, 4 = isospin-down (s)quarks
$g = 1 \dots 3$	(s)fermion generation
$s = 1 \dots 2$	number of sfermion mass-eigenstate, in the absence of mixing 1 = L, 2 = R
$c = 1 \dots 2$	number of chargino mass-eigenstate
$n = 1 \dots 4$	number of neutralino mass-eigenstate

For each block B the offset into `slhadata` and the length are respectively defined as `OffsetB` and `LengthB`. The contents of the block can be addressed through the macro `BlockB(i)`, where i runs from 1 to `LengthB`.

Matrices have a “Flat” array superimposed for convenience, in Fortran’s standard column-major convention, e.g. `USf(1,1) ≡ USfFlat(1)`, `USf(2,1) ≡ USfFlat(2)`, `USf(1,2) ≡ USfFlat(3)`, `USf(2,2) ≡ USfFlat(4)`. This makes it possible to e.g. copy such a matrix with just a single do-loop.

Block name	Array and length	Members
SPINFO	BlockSPInfo(n) LengthSPInfo	SPInfo_Severity SPInfo_NLines SPInfo_Code(n) $n = 1 \dots 15$ SPInfo_Text(i, n) do not address directly
DCINFO	BlockDCInfo(n) LengthDCInfo	DCInfo_Severity DCInfo_NLines DCInfo_Code(n) $n = 1 \dots 15$ DCInfo_Text(i, n) do not address directly
MODSEL	BlockModSel(n) LengthModSel	ModSel_Model ModSel_Content ModSel_RPV ModSel_CPV ModSel_FV ModSel_GridPts ModSel_Qmax ModSel_PDG(i) $i = 1 \dots 5$
SMINPUTS	BlockSMInputs(n) LengthSMInputs	SMInputs_AlfaMZ SMInputs_invAlfaMZ \equiv SMInputs_AlfaMZ SMInputs_GF SMInputs_AlfasMZ SMInputs_MZ SMInputs_Mf(t, g) $t = 2 \dots 4, g = 1 \dots 3$ SMInputs_MfFlat(i) $i = 1 \dots 9$ SMInputs_Me \equiv SMInputs_Mf(2,1) SMInputs_Mu \equiv SMInputs_Mf(3,1) SMInputs_Md \equiv SMInputs_Mf(4,1) SMInputs_Mmu \equiv SMInputs_Mf(2,2) SMInputs_Mc \equiv SMInputs_Mf(3,2) SMInputs_Ms \equiv SMInputs_Mf(4,2) SMInputs_Mtau \equiv SMInputs_Mf(2,3) SMInputs_Mt \equiv SMInputs_Mf(3,3) SMInputs_Mb \equiv SMInputs_Mf(4,3)

Table 1: Preprocessor variables defined in SLHA.h to access the `slhadata` array. The equivalence symbol (\equiv) indicates that the l.h.s. is just an alias for the r.h.s., not a new variable.

Block name	Array and length	Members
MINPAR	BlockMinPar(n) LengthMinPar	MinPar_Q MinPar_M0 MinPar_Lambda \equiv MinPar_M0 MinPar_M12 MinPar_Mmess \equiv MinPar_M12 MinPar_M32 \equiv MinPar_M12 MinPar_TB MinPar_signMUE MinPar_A MinPar_N5 \equiv MinPar_A MinPar_cgrav
EXTPAR	BlockExtPar(n) LengthExtPar	ExtPar_Q ExtPar_M1 ExtPar_M2 ExtPar_M3 ExtPar_Af(t) $t = 2 \dots 4$ ExtPar_Atau \equiv ExtPar_Af(2) ExtPar_At \equiv ExtPar_Af(3) ExtPar_Ab \equiv ExtPar_Af(4) ExtPar_MHu2 ExtPar_MHd2 ExtPar_MUE ExtPar_MA02 ExtPar_TB ExtPar_MSS(g, q) $g = 1 \dots 3, q = 1 \dots 5$ ExtPar_MSL(g) \equiv ExtPar_MSS($g, 1$) ExtPar_MSE(g) \equiv ExtPar_MSS($g, 2$) ExtPar_MSQ(g) \equiv ExtPar_MSS($g, 3$) ExtPar_MSU(g) \equiv ExtPar_MSS($g, 4$) ExtPar_MSD(g) \equiv ExtPar_MSS($g, 5$) ExtPar_N5(g) $g = 1 \dots 3$ ExtPar_lambda ExtPar_kappa ExtPar_Alambdab ExtPar_Akappa ExtPar_MUEeff

Table 2: Preprocessor variables defined in SLHA.h to access the slhadata array (cont'd).

Block name	Array and length	Members
MASS	BlockMass(n) LengthMass	Mass_Mf(t, g) $t = 1 \dots 4, g = 1 \dots 3$ Mass_MfFlat(i) $i = 1 \dots 12$ Mass_MSf(s, t, g) $s = 1 \dots 2, t = 1 \dots 4, g = 1 \dots 3$ Mass_MSfFlat(i) $i = 1 \dots 24$ Mass_MZ Mass_MW Mass_Mh0 Mass_MHH Mass_MAO Mass_MHp Mass_MH1 \equiv Mass_Mh0 Mass_MH2 \equiv Mass_MHH Mass_MH3 Mass_MA1 \equiv Mass_MAO Mass_MA2 Mass_MNeu(n) $n = 1 \dots 5$ Mass_MCha(c) $c = 1 \dots 2$ Mass_MG1 Mass_MGrav
NMIX	BlockNMix(n) LengthNMix	NMix_ZNeu(n_1, n_2) $n_1, n_2 = 1 \dots 4$ NMix_ZNeuFlat(i) $i = 1 \dots 16$
UMIX	BlockUMix(n) LengthUMix	UMix_UCha(c_1, c_2) $c_1, c_2 = 1 \dots 2$ UMix_UChaFlat(i) $i = 1 \dots 4$
VMIX	BlockVMix(n) LengthVMix	VMix_VCha(c_1, c_2) $c_1, c_2 = 1 \dots 2$ VMix_VChaFlat(i) $i = 1 \dots 4$

Table 3: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members
	BlockSfMix(n) LengthSfMix	SfMix_USf(s_1, s_2, t) $s_1, s_2 = 1 \dots 2, t = 2 \dots 4$ SfMix_USfFlat(i, t) $i = 1 \dots 4, t = 2 \dots 4$
STAUMIX	BlockStauMix(n) LengthStauMix	StauMix_USf(s_1, s_2) \equiv SfMix_USf($s_1, s_2, 2$) StauMix_USfFlat(i) \equiv SfMix_USfFlat($i, 2$)
STOPMIX	BlockStopMix(n) LengthStopMix	StopMix_USf(s_1, s_2) \equiv SfMix_USf($s_1, s_2, 3$) StopMix_USfFlat(i) \equiv SfMix_USfFlat($i, 3$)
SBOTMIX	BlockSbotMix(n) LengthSbotMix	SbotMix_USf(s_1, s_2) \equiv SfMix_USf($s_1, s_2, 4$) SbotMix_USfFlat(i) \equiv SfMix_USfFlat($i, 4$)
ALPHA	BlockAlpha(n) LengthAlpha	Alpha_Alpha
HMIX	BlockHMix(n) LengthHMix	HMix_Q HMix_MUE HMix_TB HMix_VEV HMix_MAO2
GAUGE	BlocktGauge(n) LengthGauge	Gauge_Q Gauge_g1 Gauge_g2 Gauge_g3
MSOFT	BlockMSoft(n) LengthMSoft	MSoft_Q MSoft_M1 MSoft_M2 MSoft_M3 MSoft_MHu2 MSoft_MHd2 MSoft_MSS(g, q) $g = 1 \dots 3, q = 1 \dots 5$ MSoft_MSL(g) \equiv MSoft_MSS($g, 1$) MSoft_MSE(g) \equiv MSoft_MSS($g, 2$) MSoft_MSQ(g) \equiv MSoft_MSS($g, 3$) MSoft_MSU(g) \equiv MSoft_MSS($g, 4$) MSoft_MSD(g) \equiv MSoft_MSS($g, 5$)

Table 4: Preprocessor variables defined in SLHA.h to access the slhadata array (cont'd).

Block name	Array and length	Members
	BlockAf(n) LengthAf	Af_Q(t) $t = 2 \dots 4$ Af_Af(g_1, g_2, t) $g_1, g_2 = 1 \dots 3, t = 2 \dots 4$ Af_AfFlat(i, t) $i = 1 \dots 9, t = 2 \dots 4$
AE	BlockAe(n) LengthAe	Ae_Q \equiv Af_Q(2) Ae_Af(g_1, g_2) \equiv Af_Af($g_1, g_2, 2$) Ae_AfFlat(i) \equiv Af_AfFlat($i, 2$) Ae_Atau \equiv Ae_Af(3,3)
AU	BlockAu(n) LengthAu	Au_Q \equiv Af_Q(3) Au_Af(g_1, g_2) \equiv Af_Af($g_1, g_2, 3$) Au_AfFlat(i) \equiv Af_AfFlat($i, 3$) Au_At \equiv Au_Af(3,3)
AD	BlockAd(n) LengthAd	Ad_Q \equiv Af_Q(4) Ad_Af(g_1, g_2) \equiv Af_Af($g_1, g_2, 4$) Ad_AfFlat(i) \equiv Af_AfFlat($i, 4$) Ad_Ab \equiv Ad_Af(3,3)
	BlockYf(n) LengthYf	Yf_Q(t) $t = 2 \dots 4$ Yf_Af(g_1, g_2, t) $g_1, g_2 = 1 \dots 3, t = 2 \dots 4$ Yf_AfFlat(i, t) $i = 1 \dots 9, t = 2 \dots 4$
YE	BlockYe(n) LengthYe	Ye_Q \equiv Yf_Q(2) Ye_Yf(g_1, g_2) \equiv Yf_Yf($g_1, g_2, 2$) Ye_YfFlat(i) \equiv Yf_YfFlat($i, 2$) Ye_Atau \equiv Ye_Yf(3,3)
YU	BlockYu(n) LengthYu	Yu_Q \equiv Yf_Q(3) Yu_Yf(g_1, g_2) \equiv Yf_Yf($g_1, g_2, 3$) Yu_YfFlat(i) \equiv Yf_YfFlat($i, 3$) Yu_At \equiv Yu_Yf(3,3)
YD	BlockYd(n) LengthYd	Yd_Q \equiv Yf_Q(4) Yd_Yf(g_1, g_2) \equiv Yf_Yf($g_1, g_2, 4$) Yd_YfFlat(i) \equiv Yf_YfFlat($i, 4$) Yd_Ab \equiv Yd_Yf(3,3)

Table 5: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members
RVLAMBDAIN	BlockRVLambdaIn(n) LengthRVLambdaIn	RVLambdaIn_lambda(i, j, k) $i, j, k = 1 \dots 3$ RVLambdaIn_lambdaFlat(i) $i = 1 \dots 27$
RVLAMBDA	BlockRVLambda(n) LengthRVLambda	RVLambda_Q RVLambda_lambda(i, j, k) $i, j, k = 1 \dots 3$ RVLambda_lambdaFlat(i) $i = 1 \dots 27$
RVLAMBDAPIN	BlockRVLambdaPIn(n) LengthRVLambdaPIn	RVLambdaPIn_lambdaP(i, j, k) $i, j, k = 1 \dots 3$ RVLambdaPIn_lambdaPFlat(i) $i = 1 \dots 27$
RVLAMBDAP	BlockRVLambdaP(n) LengthRVLambdaP	RVLambdaP_Q RVLambdaP_lambdaP(i, j, k) $i, j, k = 1 \dots 3$ RVLambdaP_lambdaPFlat(i) $i = 1 \dots 27$
RVLAMBDAPPIN	BlockRVLambdaPPIn(n) LengthRVLambdaPPIn	RVLambdaPPIn_lambdaPP(i, j, k) $i, j, k = 1 \dots 3$ RVLambdaPPIn_lambdaPPFlat(i) $i = 1 \dots 27$
RVLAMBDAPP	BlockRVLambdaPP(n) LengthRVLambdaPP	RVLambdaPP_Q RVLambdaPP_lambdaPP(i, j, k) $i, j, k = 1 \dots 3$ RVLambdaPP_lambdaPPFlat(i) $i = 1 \dots 27$
RVAIN	BlockRVAIN(n) LengthRVAIN	RVAIN_A(i, j, k) $i, j, k = 1 \dots 3$ RVAIN_AFlat(i) $i = 1 \dots 27$
RVA	BlockRVA(n) LengthRVA	RVA_Q RVA_A(i, j, k) $i, j, k = 1 \dots 3$ RVA_AFlat(i) $i = 1 \dots 27$
RVAPIN	BlockRVAPIn(n) LengthRVAPIn	RVAPIn_AP(i, j, k) $i, j, k = 1 \dots 3$ RVAPIn_APFlat(i) $i = 1 \dots 27$
RVAP	BlockRVAP(n) LengthRVAP	RVAP_Q RVAP_AP(i, j, k) $i, j, k = 1 \dots 3$ RVAP_APFlat(i) $i = 1 \dots 27$
RVAPPIN	BlockRVAPPIn(n) LengthRVAPPIn	RVAPPIn_APP(i, j, k) $i, j, k = 1 \dots 3$ RVAPPIn_APPFlat(i) $i = 1 \dots 27$
RVAPP	BlockRVAPP LengthRVAPP	RVAPP_Q RVAPP_APP(i, j, k) $i, j, k = 1 \dots 3$ RVAPP_APPFlat(i) $i = 1 \dots 27$

Table 6: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members
RVKAPPAIN	BlockRVKappaIn(n) LengthRVKappaIn	RVKappaIn_kappa(i) $i = 1 \dots 3$
RVKAPPA	BlockRVKappa(n) LengthRVKappa	RVKappa_Q RVKappa_kappa(i) $i = 1 \dots 3$
RVDIN	BlockRVDIn(n) LengthRVDIn	RVDIn_D(i) $i = 1 \dots 3$
RVD	BlockRVD(n) LengthRVD	RVD_Q RVD_D(i) $i = 1 \dots 3$
RVSNEVIN	BlockRVSneVIN(n) LengthRVSneVIN	RVSneVIN_VEV(i) $i = 1 \dots 3$
RVSNEV	BlockRVSneVEV(n) LengthRVSneVEV	RVSneVEV_Q RVSneVEV_VEV(i) $i = 1 \dots 3$
RVMLH1SQIN	BlockRVMLH1SqIn(n) LengthRVMLH1SqIn	RVMLH1SqIn_MLH12(i) $i = 1 \dots 3$
RVMLH1SQ	BlockRVMLH1Sq(n) LengthRVMLH1Sq	RVMLH1Sq_Q RVMLH1Sq_MLH12(i) $i = 1 \dots 3$
RVNMIX	BlockRVNMix(n) LengthRVNMix	RVNMix_ZNeu(n_1, n_2) $n_1, n_2 = 1 \dots 7$ RVNMix_ZNeuFlat(i) $i = 1 \dots 49$
RVUMIX	BlockRVUMix(n) LengthRVUMix	RVUMix_UCha(c_1, c_2) $c_1, c_2 = 1 \dots 5$ RVUMix_UChaFlat(i) $i = 1 \dots 25$
RVVMIX	BlockRVVMix(n) LengthRVVMix	RVVMix_VCha(c_1, c_2) $c_1, c_2 = 1 \dots 5$ RVVMix_VChaFlat(i) $i = 1 \dots 25$
RVHMX	BlockRVHMix(n) LengthRVHMix	RVUMix_UH(h_1, h_2) $h_1, h_2 = 1 \dots 5$ RVUMix_UHFlat(i) $i = 1 \dots 25$
RVAMIX	BlockRVAMix(n) LengthRVAMix	RVAMix_UA(h_1, h_2) $h_1, h_2 = 1 \dots 5$ RVAMix_UAFlat(i) $i = 1 \dots 25$
RVLMIX	BlockRVLMix(n) LengthRVLMix	RVLMix_CLep(l_1, l_2) $l_1, l_2 = 1 \dots 8$ RVLMix_CLepFlat(i) $i = 1 \dots 64$

Table 7: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

Block name	Array and length	Members
VCKMINPUTS	BlockVCKMInputs(n) LengthVCKMInputs	VCKMInputs_theta12 VCKMInputs_theta23 VCKMInputs_theta13 VCKMInputs_delta13
VCKM	BlockVCKM(n) LengthVCKM	VCKM_Q VCKM_theta12 VCKM_theta23 VCKM_theta13 VCKM_delta13

Table 8: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

2.2 PDG particle identifiers

`PDG.h` defines the human-readable versions of the PDG codes listed in Table 11. These are needed e.g. to access the decay information. At run time, the subroutine `SLHAPDGName` can be used to translate a PDG code into a particle name (see Sect. 3.12).

Block name	Array and length	Members
	BlockMSS2In(n) LengthMSS2In	MSS2In_MSS2(g_1, g_2, q) $g_1, g_2 = 1 \dots 3, q = 1 \dots 5$ MSS2In_MSS2Flat(i, q) $i = 1 \dots 9, q = 1 \dots 5$
MSL2IN	BlockMSL2In(n) LengthMSL2In	MSL2In_MSL2(g_1, g_2) \equiv MSS2In_MSS2($g_1, g_2, 1$) MSL2In_MSL2Flat(i) \equiv MSS2In_MSS2Flat($i, 1$)
MSE2IN	BlockMSE2In(n) LengthMSE2In	MSE2In_MSE2(g_1, g_2) \equiv MSS2In_MSS2($g_1, g_2, 2$) MSE2In_MSE2Flat(i) \equiv MSS2In_MSS2Flat($i, 2$)
MSQ2IN	BlockMSQ2In(n) LengthMSQ2In	MSQ2In_MSQ2(g_1, g_2) \equiv MSS2In_MSS2($g_1, g_2, 3$) MSQ2In_MSQ2Flat(i) \equiv MSS2In_MSS2Flat($i, 3$)
MSU2IN	BlockMSU2In(n) LengthMSU2In	MSU2In_MSU2(g_1, g_2) \equiv MSS2In_MSS2($g_1, g_2, 4$) MSU2In_MSU2Flat(i) \equiv MSS2In_MSS2Flat($i, 4$)
MSD2IN	BlockMSD2In(n) LengthMSD2In	MSD2In_MSD2(g_1, g_2) \equiv MSS2In_MSS2($g_1, g_2, 5$) MSD2In_MSD2Flat(i) \equiv MSS2In_MSS2Flat($i, 5$)
	BlockMSS2(n) LengthMSS2	MSS2_Q(q) $q = 1 \dots 5$ MSS2_MSS2(g_1, g_2, q) $g_1, g_2 = 1 \dots 3, q = 1 \dots 5$ MSS2_MSS2Flat(i, q) $i = 1 \dots 9, q = 1 \dots 5$
MSL2	BlockMSL2(n) LengthMSL2	MSL2_Q \equiv MSS2_Q(1) MSL2_MSL2(g_1, g_2) \equiv MSS2_MSS2($g_1, g_2, 1$) MSL2_MSL2Flat(i) \equiv MSS2_MSS2Flat($i, 1$)
MSE2	BlockMSE2(n) LengthMSE2	MSE2_Q \equiv MSS2_Q(2) MSE2_MSE2(g_1, g_2) \equiv MSS2_MSS2($g_1, g_2, 2$) MSE2_MSE2Flat(i) \equiv MSS2_MSS2Flat($i, 2$)
MSQ2	BlockMSQ2(n) LengthMSQ2	MSQ2_Q \equiv MSS2_Q(3) MSQ2_MSQ2(g_1, g_2) \equiv MSS2_MSS2($g_1, g_2, 3$) MSQ2_MSQ2Flat(i) \equiv MSS2_MSS2Flat($i, 3$)
MSU2	BlockMSU2(n) LengthMSU2	MSU2_Q \equiv MSS2_Q(4) MSU2_MSU2(g_1, g_2) \equiv MSS2_MSS2($g_1, g_2, 4$) MSU2_MSU2Flat(i) \equiv MSS2_MSS2Flat($i, 4$)
MSD2	BlockMSD2(n) LengthMSD2	MSD2_Q \equiv MSS2_Q(5) MSD2_MSD2(g_1, g_2) \equiv MSS2_MSS2($g_1, g_2, 5$) MSD2_MSD2Flat(i) \equiv MSS2_MSS2Flat($i, 5$)
	BlockASfMix(n) LengthASfMix	ASfMix_UASf(s_1, s_2, t) $s_1, s_2 = 1 \dots 6, t = 1 \dots 4$ ASfMix_UASfFlat(i, t) $i = 1 \dots 36, t = 1 \dots 4$
SNMIX	BlockSnMix(n) LengthSnMix	SnMix_UASf(s_1, s_2) \equiv ASfMix_UASf($s_1, s_2, 1$) SnMix_UASfFlat(i) \equiv ASfMix_UASfFlat($i, 1$)
SLMIX	BlockSlMix(n) LengthSlMix	SlMix_UASf(s_1, s_2) \equiv ASfMix_UASf($s_1, s_2, 2$) SlMix_UASfFlat(i) \equiv ASfMix_UASfFlat($i, 2$)
USQMIX	BlockUSqMix(n) LengthUSqMix	USqMix_UASf(s_1, s_2) \equiv ASfMix_UASf($s_1, s_2, 3$) USqMix_UASfFlat(i) \equiv ASfMix_UASfFlat($i, 3$)
DSQMIX	BlockDSqMix(n) LengthDSqMix	DSqMix_UASf(s_1, s_2) \equiv ASfMix_UASf($s_1, s_2, 4$) DSqMix_UASfFlat(i) \equiv ASfMix_UASfFlat($i, 4$)

Table 9: Preprocessor variables defined in SLHA.h to access the slhadata array (cont'd).

Block name	Array and length	Members
CVHMMIX	BlockCVHMMix(n)	CVHMMix_UH(h_1, h_2) $h_1, h_2 = 1 \dots 4$
	LengthCVHMMix	CVHMMix_UHFlat(i) $i = 1 \dots 16$
NMNMMIX	BlockNMNMMix(n)	NMNMMix_ZNeu(n_1, n_2) $n_1, n_2 = 1 \dots 5$
	LengthNMNMMix	NMNMMix_ZNeuFlat(i) $i = 1 \dots 25$
NMHMMIX	BlockNMHMMix(n)	NMUMix_UH(h_1, h_2) $h_1, h_2 = 1 \dots 3$
	LengthNMHMMix	NMUMix_UHFlat(i) $i = 1 \dots 9$
NMAMIX	BlockNMAMix	NMAMix_UA(h_1, h_2) $h_1, h_2 = 1 \dots 3$
	LengthNMAMix	NMAMix_UAFlat(i) $i = 1 \dots 9$

Table 10: Preprocessor variables defined in `SLHA.h` to access the `slhadata` array (cont'd).

fermions	sfermions	bosons	gauginos
PDG_nu_e	PDG_snu_e1 PDG_snu_e2	PDG_h0	PDG_neutralino1
PDG_electron	PDG_selectron1 PDG_selectron2	PDG_HH	PDG_neutralino2
PDG_up	PDG_sup1 PDG_sup2	PDG_A0	PDG_neutralino3
PDG_down	PDG_sdown1 PDG_sdown2	PDG_Hp	PDG_neutralino4
PDG_nu_mu	PDG_snu_mu1 PDG_snu_mu2	PDG_H3	PDG_neutralino5
PDG_muon	PDG_smuon1 PDG_smuon2	PDG_A2	PDG_chargino1
PDG_charm	PDG_scharm1 PDG_scharm2	PDG_photon	PDG_chargino2
PDG_strange	PDG_sstrange1 PDG_sstrange2	PDG_Z	PDG_gluino
PDG_nu_tau	PDG_snu_tau1 PDG_snu_tau2	PDG_W	PDG_gravitino
PDG_tau	PDG_stau1 PDG_stau2	PDG_gluon	
PDG_top	PDG_stop1 PDG_stop2	PDG_graviton	
PDG_bottom	PDG_sbottom1 PDG_sbottom2		

Table 11: The PDG codes defined in `PDG.h`.

3 Routines provided by the SLHA library

The file `SLHA.h` must be included in every subroutine or function that uses SLHALib routines. It contains the necessary preprocessor definitions as well as external declarations for the SLHALib routines.

The basic data structure is the double complex array `slhadata` of length `nslhadata`. These names are hard-coded into the preprocessor definitions and may not be changed by the user. As a corollary, only one instance of the `slhadata` structure can be used in any one routine. This poses no serious limitation for most applications, however.

3.1 SLHAClear

```
subroutine SLHAClear(slhadata)
double complex slhadata(nslhadata)
```

This subroutine sets all data in the `slhadata` array given as argument to the value `invalid` (defined in `SLHA.h`). It is important that this is done before using `slhadata`, or else any kind of junk that happens to be in the memory occupied by `slhadata` will later on be interpreted as valid data.

3.2 SLHARead

```
subroutine SLHARead(error, slhadata, filename, abort)
integer error, abort
double complex slhadata(nslhadata)
character*(*) filename
```

This subroutine reads the data in SLHA format from `filename` into the `slhadata` array. If the specified file cannot be opened, the function issues an error message and returns `error = 1`. The `abort` flag governs what happens when superfluous text is read, i.e. text that cannot be interpreted as SLHA data. If `abort` is 0, a warning is printed and reading continues. Otherwise, reading stops at the offending line and `error = 2` is returned. `SLHARead` implicitly calls `SLHAClear` to clear the `slhadata` array before reading the file.

The blocks `SPINFO` and `DCINFO` are largely ignored when reading the file, as they are for human information only. Only the maximum of all message codes is kept in the `Severity` member of the block. Since the message codes increase with severity, this indicates the overall reliability of the corresponding data (spectrum or decay information). For example, if the `Severity` member is 4 (real errors), the Accord advises not to use the corresponding data. See also Sect. 3.4.

3.3 SLHAWrite

```
subroutine SLHAWrite(error, slhadata, filename)
integer error
double complex slhadata(nslhadata)
character*(*) filename
```

This subroutine writes the data in `slhadata` to `filename`.

3.4 SLHAInfo

```
subroutine SLHAInfo(slablock, code, text)
double complex slablock(*)
integer code
character*(*) text
```

This subroutine adds a message to one of the informational blocks, SPINFO or DCINFO. The block is most conveniently addressed through the `Block...` macros, for example

```
call SLHAInfo(BlockSPInfo(1), 4, "Error in computation")
```

Allowed codes are

- 1 = program name,
- 2 = program version,
- 3 = warning message,
- 4 = error message.

Messages are truncated at 80 characters.

3.5 SLHANewDecay

```
integer function SLHANewDecay(slhadata, width, parent_id)
double complex slhadata(nslhadata)
double precision width
integer parent_id
```

This function initiates the setting of decay information for the particle specified by the `parent_id` PDG code, whose total decay width is given by `width`. The return value is an integer index which is needed to subsequently add individual decay modes with `SLHAAddDecay`. If the fixed-length array `slhadata` becomes full, a warning is printed and zero is returned. If a decay of the given particle is already present in `slhadata`, it is first removed.

3.6 SLHAFindDecay

```
integer function SLHAFindDecay(slhadata, parent_id)
double complex slhadata(nslhadata)
integer parent_id
```

This function also initiates the setting of decay information. Unlike `SLHANewDecay`, it requires that the decay of the `parent_id` particle exist and reshuffles the decay information in `slhadata` such that new channels can be added to this decay. If no decay matching `parent_id` is found, the return value is 0, otherwise it is the index needed to add decay modes with `SLHAAddDecay`.

3.7 SLHAAddDecay

```
subroutine SLHAAddDecay(slhadata, br, decay,
&   nchildren, child1_id, child2_id, child3_id, child4_id)
double complex slhadata(nslhadata)
double precision br
integer decay
integer nchildren, child1_id, child2_id, child3_id, child4_id
```

This subroutine adds the decay mode

(parent_id) → child1_id child2_id child3_id child4_id

to the decay section previously initiated by `SLHANewDecay` or `SLHAFindDecay`. `decay` is the index obtained from the latter (which also set the `parent_id`) and `childn_id` are the PDG codes of the final-state particles. The branching ratio is given in `br`. If the fixed-length array `slhadata` becomes full, a warning is printed and `decay` is set to zero.

If `decay` is zero, an overflow of `slhadata` in an earlier invocation is silently assumed and no action is performed. It is therefore sufficient to check for overflow only once, after setting all decay modes (unless, of course, one needs to pinpoint the exact location of the overflow).

As with `SLHAGetDecay` (see Sect. 3.8), only the first `nchildren` of the `childn_id` are actually accessed and Fortran allows to omit the remaining ones in the invocation.

3.8 SLHAGetDecay

```
double precision function SLHAGetDecay(slhadata, parent_id,
&   nchildren, child1_id, child2_id, child3_id, child4_id)
double complex slhadata(*)
integer parent_id
integer nchildren, child1_id, child2_id, child3_id, child4_id
```

This function extracts the decay

$$\text{parent_id} \rightarrow \text{child1_id} \text{ child2_id} \text{ child3_id} \text{ child4_id}$$

from the `slhadata` array, or the value `invalid` (defined in `SLHA.h`) if no such decay can be found. The parent and child particles are given by their PDG identifiers (see Sect. 2.2). The return value is the total decay width if `nchildren = 0`, otherwise the branching ratio of the specified channel.

Note that only the first `nchildren` of the `childn_id` are actually accessed and Fortran allows to omit the remaining ones in the invocation (a strict syntax checker might issue a warning, though). Thus, for instance,

```
Zbb = SLHAGetDecay(slhadata, PDG_Z, 2, PDG_bottom, -PDG_bottom)
```

is a perfectly legitimate way to extract the $Z \rightarrow b\bar{b}$ decay.

3.9 SLHADecayTable

```
integer function SLHADecayTable(slhadata, parent_id,
&    width, id, maxparticles, maxchannels)
double complex slhadata(nslhadata)
integer parent_id, maxparticles, maxchannels
double precision width(maxchannels)
integer id(0:maxparticles,maxchannels)
```

This function stores all decay channels for the particle identified by `parent_id` in the arrays `id` and `width`. Unlike `SLHAGetDecay`, one does not need to know the exact decay mode in order to extract information. The value 0 for `parent_id` serves as a wildcard and transfers the entire decay table contained in `slhadata`. `SLHADecayTable` returns the number of channels found. The two arrays can be read out rather straightforwardly:

For each channel c ,

- $n = \text{id}(0, c)$ gives the number of participating particles, i.e. the number of decay products plus one.
- The PDG code of the decaying particle is in `id(1, c)`.
- The PDG codes of the decay products are in `id(2, c) ... id(n, c)`.
- If $n = 1$, `width(c)` contains the decaying particle's total width in GeV.
- If $n > 1$, `width(c)` contains the branching ratio for the given decay.

3.10 SLHAExist

```
integer function SLHAExist(slablock, length)
double complex slablock(*)
integer length
```

This function tests whether a given SLHA block is not entirely empty. It returns 2 if the block has at least one complex member, 1 if the block has at least one real member (i.e. all imaginary parts zero), and 0 if the block has no valid members at all. The SLHA blocks are most conveniently accessed using the `Block...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHAExist(BlockMass(1), LengthMass) .ne. 0 ) ...
```

3.11 SLHValid

```
logical function SLHValid(slablock, length)
double complex slablock(*)
integer length
```

This function tests whether a given SLHA block consists entirely of valid data, i.e. it returns `.FALSE.` if at least one member of the block is invalid. The SLHA blocks are most conveniently accessed using the `Block...` and `Length...` definitions (see Sect. 2), e.g.

```
if( SLHValid(BlockNMix(1), LengthNMix) ) ...
```

3.12 SLHAPDGName

```
subroutine SLHAPDGName(code, name)
integer code
character*(PDGLen) name
```

This subroutine translates a PDG code into a particle name. The sign of the PDG code is ignored, hence the same name is returned for a particle and its antiparticle. The maximum length of the name, `PDGLen`, is defined in `PDG.h`.

3.13 Incompatible Changes

Two incompatible changes in the interface were necessary with respect to the SLHALib 1 [3], largely due to the fact that the SLHA2 allows complex entries:

- `slhadata` is now a double complex, not a double precision array.

- The `SLHAExist` function has become an integer function, as it now distinguishes *three* possible scenarios: no valid entries, only real entries, and complex entries.
- The `SLHAWrite` subroutine no longer has arguments for program name and version. Such informational messages can now be added with the much more general subroutine `SLHAInfo`.

4 Examples

Consider the following example program, which just copies one SLHA file to another:

```

program copy_slha_file
  implicit none

#include "SLHA.h"

  integer error
  double complex slhadata(nslhadata)

  call SLHARead(error, slhadata, "infile.slha", 0)
  if( error .ne. 0 ) stop "Read error"

  call SLHAInfo(BlockSPInfo(1), 1, "My Test Program")
  call SLHAInfo(BlockSPInfo(1), 2, "1.0")

  call SLHAWrite(error, slhadata, "outfile.slha")
  if( error .ne. 0 ) stop "Write error"
end

```

Already in this simple program a couple of things can be seen:

- the file `SLHA.h` must be included in every function or subroutine that uses the SLHA routines and this must be done using the preprocessor `#include` (not Fortran's `include`), thus the program file should have the extension `.F` (capital F).
- `slhadata` must be declared as a double complex array of length `nslhadata`.
- One should not continue with processing if a non-zero error flag is returned.

A more sensible application would add something to the `slhadata` before writing them out again. The next little program pretends to compute the fermionic Z decays (by calling a hypothetical subroutine `MyCalculation`) and adds them to `slhadata`:

```

        program compute_decays
        implicit none

#include "SLHA.h"
#include "PDG.h"

        integer error, decay, t, g
        double complex slhadata(nslhadata)
        double precision total_width, br(4,3)
        integer ferm_id(4,3)
        data ferm_id /
&    PDG_nu_e, PDG_electron, PDG_up, PDG_down,
&    PDG_nu_mu, PDG_muon, PDG_charm, PDG_strange,
&    PDG_nu_tau, PDG_tau, PDG_top, PDG_bottom /

        call SLHARead(error, slhadata, "infile.slha", 0)
        if( error .ne. 0 ) stop "Read error"

* compute the decays with parameters taken from the slhadata:
        call MyCalculation(SMInputs_MZ, MinPar_TB, ...,
&    total_width, br)

        decay = SLHANewDecay(slhadata, total_width, PDG_Z)
        do g = 1, 3
            do t = 1, 4
                call SLHAAddDecay(slhadata, br(t,g), decay,
&    2, ferm_id(t,g), -ferm_id(t,g))
            enddo
        enddo

        call SLHAInfo(BlockDCInfo(1), 1, "My Decay Calculator")
        call SLHAInfo(BlockDCInfo(1), 2, "3.1415")

        call SLHAWrite(error, slhadata, "outfile.slha")
        if( error .ne. 0 ) stop "Write error"
        end

```

Demonstrated here is the access of SLHA data (SMInputs_MZ, MinPar_TB) and the setting of decay information.

5 Building and Compiling

The SLHA library package can be downloaded as a gzipped tar archive from the Web site <http://www.feynarts.de/slha>. After unpacking the archive, change into the directory SLHALib-2.0 and type

```
./configure
make
```

Some simple demonstration programs (in the `demo` subdirectory) are built together with the library `libSLHA.a`.

Compiling a program that uses the SLHA library is in principle equally straightforward. The only tricky thing is that one has to relax Fortran's 72-column limit. This is because even lines perfectly within the 72-column range may become longer after the preprocessor's substitutions. While essentially every Fortran compiler offers such an option, the name is quite different. A glance at the man page should suffice to find out. Here are a few common choices:

Compiler	Platform/OS	Option name
g77	any	<code>-ffixed-line-length-none</code>
pgf77	Linux x86	<code>-Mextend</code>
ifort	Linux x86	<code>-extend_source</code>
f77	Tru64 Alpha	<code>-extend_source</code>
f77	SunOS, Solaris	<code>-e</code>
fort77	HP-UX	<code>+es</code>

To compile and link your program, add this option and `-Ipath -Lpath -lSLHA` to the compiler command line, where *path* is the location of the SLHA library, e.g.

```
pgf77 -Mextend -I$HOME/SLHALib-2.0 myprogram.F -L$HOME/SLHALib-2.0 -lSLHA
```

All externally visible symbols of the SLHA library start with the prefix `SLHA` and should thus pretty much avoid symbol conflicts.

6 Summary

The SLHA library presented here provides simple functions to read and write files in SLHA format. Data are kept in a single double complex array and accessed through preprocessor variables. The library is written in native Fortran 77 and is easy to build. The source code is openly available at <http://www.feynarts.de/slha> and is distributed under the GNU Library General Public License.

The author welcomes any kind of feedback, in particular bug and performance reports, at hahn@feynarts.de.

Acknowledgements

I thank J. Guasch for providing a first implementation of some SLHA2 features and introducing useful routines.

References

- [1] P. Skands et al., hep-ph/0311123.
- [2] B.C. Allanach et al., hep-ph/0602198.
- [3] T. Hahn, hep-ph/0408283.
- [4] T. Hahn and C. Schappacher, *Comp. Phys. Commun.* **143** (2002) 54 [hep-ph/0105349].